Oblivious Neural Network Predictions via MiniONN transformations

Jian Liu
Aalto University
jian.liu@aalto.fi

Yao Lu
Aalto University
yao.lu@aalto.fi

Mika Juuti
Aalto University
mika.juuti@aalto.fi

N. Asokan
Aalto University
asokan@acm.org

ABSTRACT

Machine learning models hosted in a cloud service are increasingly popular but risk privacy: clients sending prediction requests to the service need to disclose potentially sensitive information. In this paper, we explore the problem of privacy-preserving predictions: after each prediction, the server learns nothing about clients’ input and clients learn nothing about the model.

We present MiniONN, the first approach for transforming an existing neural network to an oblivious neural network supporting privacy-preserving predictions with reasonable efficiency. Unlike prior work, MiniONN requires no change to how models are trained. To this end, we design oblivious protocols for commonly used operations in neural network prediction models. We show that MiniONN outperforms existing work in terms of response latency and message sizes. We demonstrate the wide applicability of MiniONN by transforming several typical neural network models trained from standard datasets.

CCS CONCEPTS

- Security and privacy → Privacy-preserving protocols;

KEYWORDS

privacy, machine learning, neural network predictions

1 INTRODUCTION

Machine learning is now used extensively in many application domains such as pattern recognition [10], medical diagnosis [24] and credit-risk assessment [3]. Applications of supervised machine learning methods have a common two-phase paradigm: (1) a training phase in which a model is trained from some training data, and (2) a prediction phase in which the trained model is used to predict categories (classification) or continuous values (regression) given some input data. Recently, a particular machine learning framework, neural networks (sometimes referred to as deep learning), has gained much popularity due to its record-breaking performance in many tasks such as image classification [36], speech recognition [19] and complex board games [34].

Machine learning as a service (MLaaS) is a new service paradigm that uses cloud infrastructures to train models and offer online prediction services to clients. While cloud-based prediction services have clear benefits, they put clients’ privacy at risk because the input data that clients submit to the cloud service may contain sensitive information. A naive solution is to have clients download the model and run the prediction phase on client-side. However, this solution has several drawbacks: (1) it becomes more difficult for service providers to update their models; (2) the trained model may constitute a competitive advantage and thus requires confidentiality; (3) for security applications (e.g., spam or malware detection services), an adversary can use the model as an oracle to develop strategies for evading detection; and (4) if the training data contains sensitive information (such as patient records from a hospital) revealing the model may compromise privacy of the training data or even violate regulations like the Health Insurance Portability and Accountability Act of 1996 (HIPAA).

A natural question to ask is, given a model, whether is it possible to make it oblivious: it can compute predictions in such a way that the server learns nothing about clients’ input, and clients learn nothing about the model except the prediction results. For general machine learning models, nearly practical solutions have been proposed [6, 13, 14, 56]. However, privacy-preserving deep learning prediction models, which we call oblivious neural networks (ONN), have not been studied adequately. Gilad-Bachrach et al. [27] proposed using a specific activation function (“square”) and pooling operation (mean pooling) during training so that the resulting model can be made oblivious using their CryptoNets framework. CryptoNets transformations result in reasonable accuracy but incur high performance overhead. Very recently, Mohassel and Zhang [43] also proposed new activation functions that can be efficiently computed by cryptographic techniques, and use them in the training phase of their SecureML framework. What is common to both approaches [27, 43] is that they require changes to the training phase and thus are not applicable to the problem of making existing neural models oblivious.

In this paper, we present MiniONN (pronounced minion), a practical ONN transformation technique to convert any given neural network model (trained with commonly used operations) to an ONN. We design oblivious protocols for operations routinely used by neural network designers: linear transformations, popular activation functions and pooling operations. In particular, we use polynomial splines to approximate nonlinear functions (e.g., sigmoid and tanh) with negligible loss in prediction accuracy. None of our protocols require any changes to the training phase of the model being transformed. We only use lightweight cryptographic primitives such as secret sharing and garbled circuits in online prediction phase. We also introduce an offline precomputation phase to perform request-independent operations using additively homomorphic encryption together with the SIMD batch processing technique.
Our contributions are summarized as follows:

- We present MiniONN, the first technique that can transform any common neural network model into an oblivious neural network without any modifications to the training phase (Section 4).
- We design oblivious protocols for common operations in neural network predictions (Section 5). In particular, we make nonlinear functions (e.g., sigmoid and tanh) amenable for our ONN transformation with a negligible loss in accuracy (Section 5.3.2).
- We build a full implementation of MiniONN and demonstrate its wide applicability by using it to transform neural network models trained from several standard datasets (Section 6). In particular, for the same models trained from the MNIST dataset [37], MiniONN performs significantly better than previous work [27, 43] (Section 6.1).
- We analyze how model complexity impacts both prediction accuracy and computation/communication overhead of the transformed ONN. We discuss how a neural network designer can choose the right tradeoff between prediction accuracy and overhead. (Section 7).

2 BACKGROUND AND PRELIMINARIES

We now introduce the machine learning and cryptographic preliminaries (notation we use is summarized in Table 1).

<table>
<thead>
<tr>
<th>( \mathcal{S} )</th>
<th>( \mathcal{C} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>Client</td>
</tr>
<tr>
<td>( X = [x_1, \ldots] )</td>
<td>Input matrix for each layer</td>
</tr>
<tr>
<td>( W = [w_1, \ldots] )</td>
<td>Weight matrix for each layer</td>
</tr>
<tr>
<td>( B = [b_1, \ldots] )</td>
<td>Bias matrix for each layer</td>
</tr>
<tr>
<td>( Y = [y_1, \ldots] )</td>
<td>Output matrix for each layer</td>
</tr>
<tr>
<td>( z = [z_1, \ldots] )</td>
<td>Final predictions</td>
</tr>
</tbody>
</table>

\( E \)’s share of the dot-product triple
\( C \)’s share of the dot-product triple
\( Z \)’s plaintext space
\( \text{compare}(x, y) \) return 1 if \( x \geq y \), return 0 if \( x < y \)
\( E(\cdot) / \mathcal{D}(\cdot) \) Additively homomorphic encryption/decryption
\( \mathcal{P} / \mathcal{S} \) Public/Private key
\( \otimes \) Addition between two ciphertexts
\( \oplus \) Subtraction between two ciphertexts
\( \odot \) Multiplication between a plaintext and a ciphertext

Table 1: Notation table.

2.1 Neural networks

A neural network consists of a pipeline of layers. Each layer receives input and processes it to produce an output that serves as input to the next layer. Conventionally, layers are organized so that the bottom-most layer receives input data (e.g., an image or a word) and the top-most layer outputs the final predictions. A typical neural network\(^1\) processes input data in groups of layers, by first applying linear transformations, followed by the application of a nonlinear activation function. Sometimes a pooling operation is included to aggregate groups of inputs.

We will now briefly describe these operations from the perspective of transforming neural networks to ONNs.

2.1.1 Linear transformations. The commonest linear transformations in neural networks are matrix multiplications and additions:

\[
y := W \cdot x + b, \tag{1}
\]

where \( x \in \mathbb{R}^{1 \times 1} \) is the input vector, \( y \in \mathbb{R}^{n \times 1} \) is the output, \( W \in \mathbb{R}^{n \times 1} \) is the weight matrix and \( b \in \mathbb{R}^{n \times 1} \) is the bias vector.

Convolution is a type of linear transformation, which computes the dot product of small “weight tensors” (filters) and the neighborhood of an element in the input. The process is repeated, by sliding each filter by a certain amount in each step. The size of the neighborhood is called window size. The step size is called stride. In practice, for efficiency reasons, convolution is converted into matrix multiplication and addition as well [17], similar to equation 1, except that input and bias vector are matrices: \( Y := W \cdot X + B \).

The commonest linear transformations are matrix multiplications and addition.

2.1.2 Activation functions. Neural networks use nonlinear transformations of data – activation functions – to model nonlinear relationships between input data and output predictions. We identify three common categories:

- *Piecewise linear activation functions*. This category of functions can be represented as a set of \( n \) linear functions within specific ranges, each of the type \( f_i(y) = a_i y + b_i, y \in [y_i, y_{i+1}] \), where \( y_i \) and \( y_{i+1} \) are the lower and upper bounds for the range. This category includes the activation functions:
  - *Identity function (linear)*: \( f(y) = y \)
  - *Rectified Linear Units (ReLU)*: \( f(y) = \max(0, y_i) \)
  - *Leaky ReLU*: \( f(y) = \max(0, y_i) + \alpha \min(0, y_i) \)
  - *Maxout (n pieces)*: \( f(y) = \max(y_{i_1}, \ldots, y_{n}) \)

- *Smooth activation functions*. A smooth function has continuous derivatives up to some desired order over some domain. Some commonly used smooth activation functions are:
  - *Sigmoid (logistic)*: \( f(y) = \frac{1}{1 + \exp(-y)} \)
  - *Hyperbolic tangent (tanh)*: \( f(y) = \frac{e^{y_i} - 1}{e^{y_i} + 1} \)
  - *Softplus*: \( f(y) = \log(\exp(y_i) + 1) \)

The sigmoid and tanh functions are closely related [29]:

\[
tanh(x) = 2 \cdot \text{sigmoid}(2x) - 1. \tag{2}
\]

They are collectively referred to as *sigmoidal functions*.

- *Softmax*. Softmax is defined as:

\[
f(y) = \frac{e^{y_i}}{\sum_i e^{y_i}}
\]

It is usually applied to the last layer to compute a probability distribution in categorical classification. However, in prediction
phase, usually it is sufficient to use argmax over the outputs of the last layer to predict the most likely outcome.

2.1.3 Pooling operations. Neural networks also commonly use pooling operations that arrange input into several groups and aggregate inputs within each group. Pooling is commonly done by calculating the average or the maximum value among the inputs (mean or max pooling). Convolution and pooling operations are only used if the input data has spatial structure (e.g., images, sounds).

2.1.4 Commonly used neural network operations. As discussed in Section 2.1.1, all common linear transformations reduce to matrix multiplications and additions in the prediction phase. Therefore it is sufficient for an ONN transformation technique to support making matrix multiplications and additions oblivious.

To get an idea of commonly used activation functions, consider five top performing neural networks in the MNIST [37] and CIFAR-10 [35] datasets. Collectively they support the following activation functions: ReLU [38, 49, 55], leaky ReLU [31, 53], maxout [16, 42] and tanh [18]. In addition, sigmoidal activation functions are commonly used in language modeling. Finally, as we saw in Section 2.1.3 common pooling operations are mean and max pooling.

We thus argue that for an ONN transformation technique to be useful in practice, it should support all of the above commonly used neural network operations. We describe these in Sections 3 to 5.

Note that although softmax is a popular operation used in the last layer, it can be left out of an ONN [27] (e.g., the input to the softmax layer can be returned to the client) because its application is order-preserving and thus will not change the prediction result.

2.2 Cryptographic preliminaries

2.2.1 Secure two-party computation. Secure two-party computation (2PC) is a type of protocols that allow two parties to jointly compute a function \((f_1(x, y), f_2(x, y)) \rightarrow F(x, y)\) without learning each other’s input. It offers the same security guarantee achieved by a trusted third party TTP running \(F\): both parties submit their inputs (i.e., \(x, y\)) to TTP, who computes and returns the corresponding output to each party, so that no information has been leaked except the information that can be inferred from the outputs. Basically, there are three techniques to achieve 2PC: arithmetic secret sharing [8], boolean secret sharing [28] and Yao’s garbled circuits [37, 58]. Each technique has its pros and cons, and they can be converted among each other. The ABY framework [20] is a state-of-the-art 2PC library that implements all three techniques.

2.2.2 Homomorphic encryption. A public key encryption scheme is additively homomorphic if given two ciphertexts \(\hat{x}_1 := E(pk, x_1)\) and \(\hat{x}_2 := E(pk, x_2)\), there is a public-key operation \(\oplus\) such that \(E(pk, x_1 + x_2) \leftrightarrow \hat{x}_1 \oplus \hat{x}_2\). Examples of such schemes are Paillier’s encryption [47], and exponential ElGamal encryption [23]. This kind of encryption schemes is simply referred to as homomorphic encryption (HE).

As an inverse of addition, subtraction \(\ominus\) is trivially supported by additively homomorphic encryption. Furthermore, adding or multiplying a ciphertext by a constant is efficiently supported: \(E(pk, a + x) \leftrightarrow a \oplus \hat{x}\) and \(E(pk, a \cdot x_1) \leftrightarrow a \ominus \hat{x}_1\).

To do both addition and multiplication between two ciphertexts, fully homomorphic encryption (FHE) or leveled homomorphic encryption (LHE) is needed. However, FHE requires expensive bootstrapping operations and LHE only supports a limited number of homomorphic operations.

2.2.3 Single instruction multiple data (SIMD). The ciphertext of a (homomorphic) encryption scheme is usually much larger than the data being encrypted, and the homomorphic operations on the ciphertexts take longer time than those on the plaintexts. One way to alleviate this issue is to encode several messages into a single plaintext and use the single instruction multiple data (SIMD) [52] technique to process these encrypted messages in batch without introducing any extra cost. The LHE library [22] has implemented SIMD based on the Chinese Reminder Theorem (CRT). In this paper, we use \(\bar{x}\) to denote the encryption of a vector \([x_1, \ldots, x_n]\) in batch using the SIMD technique.

The SIMD technique can also be applied to secure two-party computation to reduce the memory footprint of the circuit and improve the circuit evaluation time [11]. In traditional garbled circuits, each wire stores a single input, while in the SIMD version, an input is split across multiple wires so that each wire corresponds to multiple inputs. The ABY framework [20] supports this.

3 PROBLEM STATEMENT

We consider the generic setting for cloud-based prediction services, where a server \(S\) holds a neural network model, and clients \(C\)'s submit their input to learn corresponding predictions. The model is defined as:

\[
z := (W_L \cdot f_{L-1}(...f_1(W_1 \cdot X + b_1)...) + b_L)
\]

The problem we tackle is how to design oblivious neural networks: after each prediction, \(S\) learns nothing about \(X\), and \(C\) learns nothing about \((W_1, W_2, ..., W_L)\) and \((b_1, b_2, ..., b_L)\) except \(z\). Our security definition follows the standard ideal-world/real-world paradigm: the adversary’s view in real-world is indistinguishable to that in ideal-world.

Adversary model. We assume that either \(S\) or \(C\) can be compromised by an adversary \(A\), but not at the same time. We assume \(A\) to be semi-honest, i.e., it directs the corrupted party to follow the protocol specification in real-world, and submits the inputs it received from the environment to TTP in ideal-world. We rely on efficient implementations of primitives (like 2PC in ABY framework [20]) that are secure against semi-honest adversaries.

A compromised \(S\) tries to learn the values in \(X\), and a compromised \(C\) tries to learn the values in \(W\) and \(B\). We do not aim to protect the sizes of \(X, W, B\), and which \(f()\) is being used. However, \(S\) can protect such information by adding dummy layers. Note that \(C\) can, in principle, use \(S\)'s prediction service as a blackbox oracle to extract an equivalent or near-equivalent model (model extraction attacks [54]), or even infer the training set (model inversion [25] or membership inference attacks [51]). However, in a client-server setting, \(S\) can rate limit prediction requests from a given \(C\), thereby slowing down or bounding this information leakage.
4 MINIONN OVERVIEW

In this section, we explain the basic idea of MINIONN by transforming a toy neural network of the form:

\[ z := W' \cdot f(W \cdot x + b) + b' \quad (4) \]

where \( x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \ W = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}, \ b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \ W' = \begin{bmatrix} w'_1,1 & w'_1,2 \\ w'_2,1 & w'_2,2 \end{bmatrix} \)

and \( b' = \begin{bmatrix} b'_1 \\ b'_2 \end{bmatrix} \).

The core idea of MINIONN is to have \( S \) and \( C \) additively share each of the input and output values for every layer of a neural network. That is, at the beginning of every layer, \( S \) and \( C \) will each hold a “share” such that modulo addition of the shares is equal to the input to that layer in the non-oblivious version of that neural network. The output values will be used as inputs for the next layer.

To this end, we have \( S \) and \( C \) first engage in a precomputation phase (which is independent of \( C \)’s input \( x \)), where they jointly generate a set of dot-product triplets \((u, v, w, r)\) for each row of the weight matrices \((\mathbf{W}, \mathbf{W}')\) in this example. Specifically, for each row \( w \), \( S \) and \( C \) run a protocol that securely implements the ideal functionality \( F_{\text{triplet}} \) (in Figure 1) to generate dot-product triplets, such that:

\[
\begin{align*}
    u_1 + v_1 \pmod{N} &= w_{1,1}r_1 + w_{1,2}r_2, \\
    u_2 + v_2 \pmod{N} &= w_{2,1}r_1 + w_{2,2}r_2, \\
    u'_1 + v'_1 \pmod{N} &= w'_{1,1}r'_1 + w'_{1,2}r'_2, \\
    u'_2 + v'_2 \pmod{N} &= w'_{2,1}r'_1 + w'_{2,2}r'_2.
\end{align*}
\]

Figure 1: Ideal functionality \( F_{\text{triplet}} \): generate a dot-product triplet.

When \( C \) wants to ask \( S \) to compute the predictions for a vector \( x = [x_1, x_2] \), for each \( x_i \), \( C \) chooses a triplet generated in the precomputation phases and uses its \( r_i \) value to blind \( x_i \):

\[
\begin{align*}
    x'_1 &= r_1, \quad x'_1 = x_1 - r_1 \pmod{N}, \\
    x'_2 &= r_2, \quad x'_2 = x_2 - r_2 \pmod{N},
\end{align*}
\]

\( C \) then sends \( x^S \) to \( S \), who calculates

\[
\begin{align*}
    y^S_1 &= w_{1,1}x'_1^S + w_{1,2}x'_2^S + b_1 + u_1 \pmod{N}, \\
    y^S_2 &= w_{2,1}x'_1^S + w_{2,2}x'_2^S + b_2 + u_2 \pmod{N}.
\end{align*}
\]

Meanwhile, \( C \) sets:

\[
\begin{align*}
    y^C_1 &= v_1 \pmod{N}, \\
    y^C_2 &= v_2 \pmod{N}.
\end{align*}
\]

It is clear that

\[
\begin{align*}
    y^C_1 + y^S_1 \pmod{N} &= w_{1,1}x_1 + w_{1,2}x_2 + b_1 \quad \text{and} \\
    y^C_2 + y^S_2 \pmod{N} &= w_{2,1}x_1 + w_{2,2}x_2 + b_2.
\end{align*}
\]

Therefore, at the end of this interaction, \( S \) and \( C \) additively share the output values \( y \) resulting from the linear transformation in layer 1 without \( S \) learning the input \( x \) and neither party learning \( y \).

In Section 5.2 we describe the detailed operations for making linear transformations oblivious.

For the activation/pooling operation \( f() \), \( S \) and \( C \) run a protocol that securely implements the ideal functionality in Figure 2, which implicitly reconstructs each \( u_i := y^C_i + y^S_i \pmod{N} \) and returns \( x'_2 := f(y_i) - x'_2 \pmod{N} \) to \( S \), where \( x'_2 \) is \( C \)'s component of a previously shared triplet from the precomputation phase, i.e., \( x'_1 := r'_1 \) and \( x'_2 := r'_2 \). In Sections 5.3 and 5.4, we show how the ideal functionality in Figure 2 can be concretely realized for commonly used activation functions and pooling operations.

Figure 2: Ideal functionality: oblivious activation/pooling \( f() \).

The transformation of the final layer is the same as the first layer. Namely, \( S \) calculates:

\[
\begin{align*}
    y^S_1 &= w'_{1,1}x'_1^S + w'_{1,2}x'_2^S + b'_1 + u'_1 \pmod{N}, \\
    y^S_2 &= w'_{2,1}x'_1^S + w'_{2,2}x'_2^S + b'_2 + u'_2 \pmod{N};
\end{align*}
\]

and \( C \) sets:

\[
\begin{align*}
    y^C_1 &= v'_1 \pmod{N}, \\
    y^C_2 &= v'_2 \pmod{N}.
\end{align*}
\]

At the end, \( S \) returns \([y^S_1, y^S_2]\) back to \( C \), who outputs the final predictions:

\[
\begin{align*}
    z_1 &= y^C_1 + y^S_1, \\
    z_2 &= y^C_2 + y^S_2.
\end{align*}
\]

Note that MINIONN works in \( \mathbb{Z}_N \), while neural networks require floating-point calculations. A simple solution is to scale the floating-point numbers up to integers by multiplying the same constant to all values and drop the fractional parts. A similar technique is used to reduce memory requirements in neural network predictions, at negligible loss of accuracy [41]. We must make sure that the absolute value of any (intermediate) results will not exceed \([N/2]\).

5 MINIONN DESIGN

5.1 Dot-product triplet generation

Recall that we introduce a precomputation phase to generate dot-product triplets, which are similar to the multiplication triplets used in secure computations [8]. Multiplication triplets are typically generated in two ways: using homomorphic encryption (HE-based) or using oblivious transfer (OT-based). The former is efficient in terms of communication, whereas the latter is efficient in terms of computation. Both approaches can be optimized for the dot-product generation [43]. In the HE-based approach, dot-products can be calculated directly on ciphertexts, so that both communication and decryption time can be reduced.
We further improve the HE-based approach using the SIMD batch processing technique. The protocol is described in Figure 3. Using the SIMD technique, $S$ encrypts the whole vector $w$ into a single ciphertext of additively homomorphic encryption. $C$ computes $\tilde{u} \leftarrow r \odot \tilde{w} \oplus v$, where $r$ and $v$ are random vectors generated by $C$. $S$ decrypts $\tilde{u}$ and outputs the sum of $u$. Meanwhile, $C$ outputs the sum of $v$. Even though $S$ and $C$ need to generate new dot-product triplets for each prediction request, $S$ only needs to transfer $\tilde{w}$s once for all predictions. Furthermore, it can pack multiple $w$s into a single ciphertext if needed.

### Input:

$S$: $w \in \mathbb{Z}_N^n$

$C$: $r \in \mathbb{Z}_N^n$

### Output:

$S$: a random number $u \in \mathbb{Z}_N$;

$C$: $v \in \mathbb{Z}_N$, s.t., $u + v \mod N = w \cdot r$.

$\tilde{w} \leftarrow E(pk_s, w)$

$v \leftarrow \mathbb{Z}_N^n$

$\tilde{u} \leftarrow r \odot \tilde{w} \oplus v$

$u \leftarrow \Sigma(D(sk_s, \tilde{u}))$

Output $u$

Figure 3: Dot-product triplet generation.

**Theorem 1.** The protocol in Figure 3 securely implements $F_{\text{triplet}}$ in the presence of semi-honest adversaries, if $E()$ is semantically secure.

**Proof.** Our security proof follows the ideal-world/real-world paradigm: in real-world, parties interact according to the protocol specification, whereas in ideal-world, parties have access to a trusted party TTP that implements $F_{\text{triplet}}$. The executions in both worlds are coordinated by the environment $E$, who chooses the inputs to the parties and plays the role of a distinguisher between the real and ideal executions. We aim to show that the adversary’s view in real-world is indistinguishable to that in ideal-world.

**Security against a semi-honest server.** First, we prove security against a semi-honest server by constructing an ideal-world simulator $Sim$ that performs as follows:

1. receives $r$ from $E$, and sends it to TTP;
2. starts running $C$ on input $r$;
3. constructs $\tilde{w}' \leftarrow E(pk'_s, [0, \ldots, 0])$ where $pk'_s$ is randomly generated by $Sim$;
4. gives $\tilde{w}'$ to $C$;
5. outputs whatever $C$ outputs.

$C$’s view in real execution is $E(pk_s, w)$, which is computationally indistinguishable from its view in ideal execution i.e., $E(pk'_s, [0, \ldots, 0])$ due to the semantic security of $E()$. Thus, the output distribution of $E$ in real-world is computationally indistinguishable from that in ideal-world.

**5.2 Oblivious linear transformations**

Recall that when $C$ wants to request $S$ to compute predictions for an input $X$, it blinds each value of $X$ using a random value $r$ from a dot-product triplet generated earlier: $x^S := x - r \mod N$. Then, $C$ sets $X^C := R$, and sends $X^S$ to $S$. The security of the dot-product generation protocol guarantees that $S$ knows nothing about the $r$ values. Consequently, $S$ cannot get any information about $X$ from $X^S$ if all $r$s are randomly chosen by $C$ from $\mathbb{Z}_N$.

Upon receiving $X^S$, $S$ will input it to the first layer which is typically a linear transformation layer. As we discussed in Section 2.1, all linear transformations can be turned into matrix multiplications/additions: $Y = W \cdot X + B$. Figure 4 shows the oblivious linear transformation protocol. For each row of $W$ and each column of $X^S$, $S$ and $C$ jointly generate a dot-product triplet: $u + v \mod N = w \cdot x^i$. Since $X^C$ is independent of $X$, they can generate such triplets in a precomputation phase. Next, $S$ calculates $Y^S := W \cdot X^S + B + U$, and meanwhile $C$ sets $Y^C := V$. Consequently, each element of $Y^S$ and $Y^C$ satisfy:

$y^S + y^C = w \cdot x^i + b + u + v$

$= w_1(x_1 - x^C_1) + \ldots + w_l(x_l - x^C_l) + b + u + v$

$= (w_1x_1 + \ldots + w_lx_l + b) - (w_1x^C_1 + \ldots + w_lx^C_l) + u + v$

$= g$

Due to the fact that $(U, V)$ are securely generated by $F_{\text{triplet}}$, the outputs of this layer (which are the inputs to the next layer) are also randomly shared between $S$ and $C$, i.e., $Y^C = V$ and $Y^S = Y - V$ can be used as inputs for the next layer directly.

It is clear that the view of both $S$ and $C$ are identical to their views under the dot-product triple generation protocol. Therefore, the oblivious linear transformation protocol is secure if $F_{\text{triplet}}$ is securely implemented.

A linear transformation layer can also follow an activation layer or a pooling layer. So, we need to design the oblivious activation/pooling operations in a way that their outputs can be the
inputs to linear transformations: \( X^S \) and \( X^C \) s.t. \( X^S + X^C = X \) and \( X^C \) has been used to generate the dot-product triplets for the next layer. See the following sections.

### 5.3 Oblivious activation functions

In this section, we introduce the oblivious activation function which receives \( y^C \) from \( C \) and \( y^S \) from \( S \), and outputs \( x^C \) to \( C \) and \( x^S := f(y^S + y^C) - x^C \) to \( S \), where \( x^C \) is a random number generated by \( C \). Note that if the next layer is a linear transformation layer, \( x^C \) should be the random value that has been used by \( C \) to generate a dot-product triplet in the precomputation phase. On the other hand, if the next layer is a pooling layer, \( x^C \) can be generated on demand.

#### 5.3.1 Oblivious piecewise linear activation functions

Piecewise linear activation functions are widely used in image classifications due to their outstanding performance in training phase as demonstrated by Krizhevsky et al. [36]. We take ReLU as an example to illustrate how to transform piecewise linear functions into their oblivious forms. Recall that ReLU is \( f(y) = \max(0, y) \), where \( y \) is additively shared between \( S \) and \( C \). An oblivious ReLU protocol will reconstruct \( y \) and return \( \max(0, y) - x^C \) to \( S \). This is equivalent to the ideal functionality \( F_{\text{ReLU}} \) in Figure 5. Actually, we compare \( y \) with \( \frac{N}{2} \): \( y > \frac{N}{2} \) implies \( y \) is negative (recall that absolute values of all intermediate results will not exceed \( \lceil N/2 \rceil \)).

\[ F_{\text{ReLU}} \text{ can be trivially implemented by a 2PC protocol. Specifically, we use a garbled circuit to reconstruct } y \text{ and calculate } b := \text{compare}(y, 0) \text{ to determine whether } y \geq 0 \text{ or not. If } y \geq 0, \text{ it returns } y; \text{ otherwise, it returns } 0. \text{ This is achieved by multiplying } y \text{ with } b. \]

The only operations we need for oblivious ReLU are +, −, and \( \text{compare} \), all of which are supported by the 2PC library [20] we used. So both implementation and security argument are straightforward.

Oblivious leaky ReLU can be constructed in the same way as oblivious ReLU, except that \( S \) gets:

\[ x^S := \text{compare}(y, 0) \cdot y - r \pmod{N} \]

#### 5.3.2 Oblivious smooth activation functions

Unlike piecewise linear functions, it is non-trivial to make smooth functions oblivious. For example, in the sigmoid function \( f(y) = \frac{1}{1 + e^y} \), both \( e^y \) and division are expensive to be computed by 2PC protocols [48]. Furthermore, it is difficult to keep track of the floating point value of \( e^y \), especially when \( y \) is blinded. It is well-known that such functions can be approximated locally by high-degree polynomials, but oblivious protocols can only handle low-degree approximation polynomials efficiently. To this end, we adapt an approximation method that can be efficiently computed by an oblivious protocol and incurs negligible accuracy loss.

**Approximation of smooth functions.** A smooth function \( f() \) can be approximated by a set of piecewise continuous polynomials, i.e., splines [21]. The idea is to split \( f() \) into several intervals, in each of which a polynomial is used to approximate \( f() \). The polynomials are chosen such that the overall goodness of fit is maximized. The approximation method is detailed in the following steps:

1. Set the approximation range \([a_1, a_m]\), select \( n \) equally spaced samples (including \( a_1 \) and \( a_m \)). The resulting sample set is \([a_1, \ldots, a_n]\).
2. For each \( a_i \), calculate \( \beta_i := f(a_i) \).
3. Find \( m \) switchover positions (i.e., knots) for polynomials expressions:
   a. fit an initial approximation \( f \) of order \( d \) for the dataset \([a_1, \beta_i]\) using polynomial regression (without knots);
   b. select a new knot \( \tilde{a}_i \in [a_1, \ldots, a_n] \) and fit two new polynomial expressions on each side of the knot (the knot is chosen such that the overall goodness of fit is maximized);
   c. repeat (b) until the number of knots equals \( m \).

The set of knots is now \([\tilde{a}_1, \ldots, \tilde{a}_m]\). Note that \( \tilde{a}_1 = a_1 \) and \( \tilde{a}_m = a_n \).

4. Fit a smoothing spline ([21], Chapter 5) of the same order using the knots \([a_i]\) on the dataset \([a_i, \beta_i]\) and extract the polynomial expression \( P_i(x) \) in the each interval \([\tilde{a}_i, \tilde{a}_{i+1}], i \in \{1, m - 1\} \).
5. Set boundary polynomials \( P_0() \) for \( \alpha < \tilde{a}_1 \) and \( P_m() \) for \( \alpha > a_n \), which are chosen specifically for \( f() \) to closely approximate the behaviour beyond the ranges \([\alpha, \tilde{a}_n]\).

\[ x^C := \text{compare}(y, 0) \cdot a \cdot y + (1 - \text{compare}(y, 0)) \cdot y - r \pmod{N} \]
Thus, we split \( f() \) into \( m + 1 \) intervals, and each has a separate polynomial expression.\(^3\)

(6) The final approximation is:

\[
\tilde{f}(\alpha) = \begin{cases} 
    P_0(\alpha) & \text{if } \alpha < \alpha_1 \\
    P_1(\alpha) & \text{if } \alpha_1 \leq \alpha < \alpha_2 \\
    \vdots & \\
    P_{m-1}(\alpha) & \text{if } \alpha_{m-1} \leq \alpha < \alpha_m \\
    P_m(\alpha) & \text{if } \alpha \geq \alpha_m.
\end{cases}
\]

Note that any univariate monotonic functions can be fitted by above procedure.

Oblivious approximated sigmoid. We take sigmoid as an example to explain how to transform smooth activation functions into their oblivious forms. We set the polynomial degree \( d \) as 1, since linear functions (as opposed to higher-degree polynomials) are faster and less memory-consuming to be computed by 2PC. The approximated sigmoid function is as follows:

\[
\tilde{f}(y) = \begin{cases} 
0 & \text{if } y < y_1 \\
    a_1 y + b_1 & \text{if } y_1 \leq y < y_2 \\
    \vdots & \\
    a_{m-1} y + b_{m-1} & \text{if } y_{m-1} \leq y < y_m \\
    1 & \text{if } y \geq y_m.
\end{cases}
\]

We will show (in Section 6.2) that it approximates sigmoid with negligible accuracy loss.

The approximated sigmoid function (Equation 6) is in fact a piecewise linear function. So it can be transformed in the same negligible accuracy loss.

The ideal functionality for the approximated sigmoid \( \tilde{f}_{\text{sigmoid}} \) is shown in Figure 6. Correctness of this functionality follows the fact that, for \( y_i \leq y < y_{i+1} \):

\[
x = ((a_i y + b_i) - (a_{i+1} y + b_{i+1}))) + ((a_{i+1} y + b_i) - (a_{i+1} y + b_{i+1})) + \ldots + ((a_{m-1} y + b_{m-1}) - 1) + 1
\]

Figure 6: The ideal functionality \( \tilde{f}_{\text{sigmoid}} \).

Even though it is more complex than \( \tilde{f}_{\text{ReLU}} \), it can still be realized easily using the basic functionalities provided by 2PC.

In summary, we support all activation functions that are both:

1) monotonic in ranges \((\infty, 0)\) and \((0, \infty)\);\(^4\)

2) either piecewise-polynomial or approximable.

Most commonly used activation functions belong to this class, except softmax that violates the second condition. But softmax can be replaced by argmax (Section 2.1.2) or left out (Section 2.1).

5.4 Oblivious pooling operations

The pooling layer arranges the inputs into several groups and take the max or mean of the elements in each group. For mean pooling, we just have \( \mathcal{S} \) and \( \mathcal{C} \) calculate the sum of their respective shares and keep track of the divisor. For max pooling, we use garbled circuits to realize the ideal functionality \( \tilde{f}_{\text{max}} \) in Figure 7, which reconstructs each \( y_i \) and returns the largest one masked by a random number. The max function can be easily achieved by the \( \text{compare} \) function.

Figure 7: The ideal functionality \( \tilde{f}_{\text{max}} \).

Note that the oblivious \text{maxout} activation can be trivially realized by the ideal functionality \( \tilde{f}_{\text{max}} \).

5.5 Remarks

5.5.1 Oblivious square function. The square function (i.e., \( f(y) = y^2 \)) is also used as an activation function in [27, 43], because it is easier to be transformed into an oblivious form. We implement an oblivious square function by realizing the ideal functionality in Figure 8 using arithmetic secret sharing.

Input:

\[
\mathcal{S}: y^S \in \mathbb{Z}_N; \\
\mathcal{C}: y^C, r \in \mathbb{Z}_N.
\]

Output:

\[
\mathcal{S}: x^S := \text{compare}(y_1, y) \cdot (0 - (a_1 y + b_1)) += \text{compare}(y_2, y) \cdot ((a_1 y + b_1) - (a_2 y + b_2)) \ldots += \text{compare}(y_{m-1}, y) \cdot ((a_{m-1} y + b_{m-1}) - 1) + 1 \mod N, \text{ where } y = y^S + y^C \mod N; \\
\mathcal{C}: x^C := r.
\]

Figure 8: The ideal functionality \( \tilde{f}_{\text{square}} \).

5.5.2 Dealing with large numbers. Recall that we must make sure that the absolute value of any (intermediate) results will not exceed \([N/2]\). However, the data range grows exponentially with the number of multiplications, and it grows even faster when the floating-point numbers are scaled to integers. Furthermore, the SIMD

\(^3\)We apply post-processing to the polynomials to ensure they are within upper and lower bounds of the function \( f() \), and to ensure that the approximate function \( \tilde{f} \) is monotonic (if \( f() \) is).

\(^4\)This condition guarantees that our scaling technique (i.e., scale the floating-point numbers up to integers by multiplying the same constant to all values and drop the fractional part) does not change the ranking order and thus does not impact prediction accuracy. Recall that the model finally outputs a set of probabilities, one for each class. The class with maximal probability is the prediction.
We implemented 128-bit security parameter and SIMD circuits. We used YASHE [12] for homomorphic encryption. The largest possible plaintext modulus: $\frac{N}{128}$ is a reasonable encryption time and ciphertext size. Then we chose the $n$ level. The SEAL library automatically chooses a secure $q$. The YASHE encryption scheme maps plaintext messages from the ring $\mathbb{Z}[x]/(x^n + 1)$ to the ring $\mathbb{Z}_{q}[x]/(x^n + 1)$. The ciphertext modulus $q$ determines the security level. The SEAL library automatically chooses a secure $q$ given the polynomial degree $n$ (i.e., SIMD batch size). Choice of $n$ is a tradeoff between parallelism and efficiency of a single encryption. We first set $n = 4096$ so that we can encrypt 4096 elements together in a reasonable encryption time and ciphertext size. Then we chose the largest possible plaintext modulus: $N = 101,285,036,033$, which is large enough for the needed precision since we securely scale down the value when it becomes large as we discussed in Section 5.5.2.

To evaluate its performance, we ran the server-side program on a remote computer (Intel Core i5 CPU with 4 3.30 GHz cores and 16 GB memory) and the client-side program on a local desktop (Intel Core i5 CPU machine with 4 3.20 GHz cores and 8 GB memory). We used the Clocks module in C++ for time measurement and used TCPdump for bandwidth measurement. We measured response latency (including the network delay) and message sizes during the whole procedure, i.e., from the time $C$ begins to generate its request to the time it obtains the final predictions. Each experiment was repeated 5 times and we calculated the mean and standard deviation. The standard deviations in all reported results are less than 3%.

### 6.1 Comparisons with previous work

The MNIST dataset [37] consists of 70,000 black-white hand-written digit images (of size $1 \times 28 \times 28$: width and height are 28 pixels) in 10 classes. There are 60,000 training images and 10,000 test images. Since previous work use MNIST to evaluate their techniques, we use it to provide a direct comparison with prior work.

#### Neural network in SecureML [43]

We reproduced the model (Figure 10) presented in SecureML [43]. It uses multi-layer perceptron (MLP) model with square as the activation function and achieves an accuracy of 93.1% in the MNIST dataset. We improve the accuracy of this model to 97.6% by using the Limited-memory BFGS [39] optimization algorithm and batch normalization during training. We transformed this model with MiniONN and compared the results with those reported in [27].

![Figure 10: The neural network presented in SecureML [43].](image-url)
<table>
<thead>
<tr>
<th>Square/MLP/MNIST</th>
<th>Latency (s)</th>
<th>Message Sizes (MB)</th>
<th>Accuracy %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>offline</td>
<td>online</td>
<td>offline</td>
</tr>
<tr>
<td>by SecureML [43]</td>
<td>4.7</td>
<td>0.18</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>93.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>by MiniONN</td>
<td>0.9</td>
<td>0.14</td>
<td>3.8</td>
</tr>
<tr>
<td></td>
<td>97.8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Comparison: MiniONN vs. SecureML [43].

<table>
<thead>
<tr>
<th>SecureML [43]</th>
<th>MiniONN</th>
</tr>
</thead>
<tbody>
<tr>
<td># homomorphic encryptions</td>
<td>784</td>
</tr>
<tr>
<td># homomorphic multiplications</td>
<td>100,352</td>
</tr>
<tr>
<td># ciphertext transfers</td>
<td>912</td>
</tr>
<tr>
<td># homomorphic decryptions</td>
<td>128</td>
</tr>
</tbody>
</table>

Table 3: Comparison: MiniONN vs. SecureML [43], dot-product triplet generations.

Neural network in CryptoNets [27]. We reproduced the model (Figure 11) presented in CryptoNets [27]. It is a CNN model with square as the activation function as well, and uses mean pooling instead of max pooling. Due to the convolution operation, it achieves a higher accuracy of 98.95% in the MNIST dataset. We transformed this model with MiniONN and compared its performance with the results reported in CryptoNets [27]. Table 4 shows that MiniONN achieves 230-fold reduction in latency and 8-fold reduction in message sizes, without degradation in accuracy. CryptoNets uses the SIMD technique to batch different requests to achieve a throughput of 51,739 predictions per hour, but these requests must be from the same client. In scenarios where the same client sends a very large number of prediction requests and can tolerate response latency in the order of minutes, CryptoNets can achieve 6-fold throughput than MiniONN. In scenarios where each client sends only a small number of requests but needs quick responses, MiniONN decisively outperforms CryptoNets.

(1) Convolution: input image $28 \times 28$, window size $5 \times 5$, stride $(2, 2)$, number of output channels $5$. It can be converted to matrix multiplication [17]: $x^{5 \times 169} \leftarrow x^{2 \times 25}, x^{2 \times 169}$.

(2) Square Activation: squares the value of each input.

(3) Pool: combination of mean pooling and linear transformation: $x^{100 \times 401} \leftarrow x^{100 \times 40}, x^{100 \times 4}$.

(4) Square Activation: squares the value of each input.

(5) Fully Connected Layer: fully connects the incoming 100 nodes to the outgoing 10 nodes: $x^{100 \times 1} \leftarrow x^{10 \times 40}, x^{10 \times 1}$.

Figure 11: The neural network presented in CryptoNets [27].

<table>
<thead>
<tr>
<th>Square/CNN/MNIST</th>
<th>Latency (s)</th>
<th>Message Sizes (MB)</th>
<th>Accuracy %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>offline</td>
<td>online</td>
<td>offline</td>
</tr>
<tr>
<td>by CryptoNets [27]</td>
<td>0</td>
<td>297.5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>98.95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>by MiniONN</td>
<td>0.88</td>
<td>0.4</td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td>98.95</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Comparison: MiniONN vs. CryptoNets [27].

6.2 Evaluations with realistic models

As we stated in Section 2, a useful ONN transformation technique must support commonly used neural network operations. Both CryptoNets and SecureML [43] fall short on this count. In this section we discuss performance evaluations of realistic models that are built with popular neural network operations using several different standard datasets.

Handwriting recognition: MNIST. We trained and implemented another neural network (Figure 12) using the MNIST dataset, but using ReLU as the activation function. The use of ReLU with a more complex neural network increases the accuracy of the model in MNIST to 99.31%, which is close to the state-of-the-art accuracy in the MNIST dataset (99.79%)\(^6\).

Image classification: CIFAR-10. CIFAR-10 [35] is a standard dataset consisting of RGB images (of size $3 \times 32 \times 32$, 3 color channels, width and height are 32) of everyday objects in 10 classes (e.g., automobile, bird etc.). The training set has 50,000 images while the test set has 10,000 images. The neural network is detailed in Figure 13. It achieves 81.61% prediction accuracy.

Language modeling: PTB. Penn Treebank (PTB) is a standard dataset [40] for language modeling, i.e., predicting likely next words given the

\(^6\)http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html (last accessed May 9, 2017)
previous words ([44], Chapter 27). We used a preprocessed version of this dataset\(^7\), which consists of 929,000 training words, 73,000 validation words, and 82,000 test words.

**Long Short Term Memory (LSTM)** is a neural network architecture that is commonly used for language modeling [32]. Sigmoidal activation functions are typically used in such networks. We reproduced and transformed a recent LSTM model [59] following the tutorial\(^8\) in Tensorflow [1]. To the extent of our knowledge, this is the first time language modeling is performed using oblivious models, which paves the way to oblivious neural machine translation. The model is described in Figure 14.

\(^7\)http://www.fit.vutbr.cz/~imikolov/rnnlm/
\(^8\)https://www.tensorflow.org/tutorials/recurrent, accessed April 20, 2017. We used the ‘small’ model configuration.

Figure 14: The neural network trained from the PTB dataset.

We used the real sigmoid activation functions for training, but replaced them with their corresponding approximations (Section 5.3.2) for predictions. In our sigmoid approximation, we set the ranges as \([\alpha_0, \alpha_1] = [-30, 30]\) and set the polynomials beyond the ranges as 0 and 1, i.e., \(\tilde{f}(y < -30) = 0\) and \(\tilde{f}(y > 30) = 1\) as in Equation 6.\(^9\)

Unlike aforementioned image datasets, prediction quality here is measured by a loss function called cross-entropy loss\(^{[38,352]}\). The optimal number of linear pieces differs on the model structure, validation words, and \(\approx 82,000\) test words.

\(^9\)This is exactly as in the Theano deep learning framework [9], where this approximation for numerical stability.

The overhead of linear transformations of models with approximated sigmoid and tanh, evaluate over the full PTB test set.

![Figure 15: Cross-entropy loss for models with approximated sigmoid/tanh](image)

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Latency (s)</th>
<th>Message Sizes (MB)</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReLU/CNN/MNIST</td>
<td>offline: 3.58, online: 5.74</td>
<td>offline: 20.9, online: 636.6</td>
<td>99.0%</td>
</tr>
<tr>
<td>ReLU/CNN/CIFAR-10</td>
<td>offline: 472, online: 72</td>
<td>offline: 3046, online: 6226</td>
<td>81.61%</td>
</tr>
<tr>
<td>Sigmoidal/LSTM/PTB</td>
<td>offline: 13.9, online: 4.39</td>
<td>offline: 86.7, online: 474</td>
<td>cross-entropy loss: 4.79</td>
</tr>
</tbody>
</table>

Table 5: Performance of MiniONN transformations of models with common activation functions and pooling operations.

### 7 Complexity, Accuracy and Overhead

In Section 6, we demonstrated that, unlike prior work, MiniONN can transform existing neural networks into oblivious variants. However, by simplifying the neural network model a designer can trade off a small sacrifice in prediction accuracy with a large reduction in the overhead associated with the ONN.

The relationship between model complexity and prediction accuracy is well-known ([29], Chapter 6). In neural networks, model complexity depends on the network structure: the number of neurons (size of output from each layer), types of operations (e.g., choice of activation functions) and the number of layers in the network. While prediction accuracy can increase with model complexity, it eventually saturates with some level of complexity.

**Model complexity vs. prediction overhead.** The overhead of linear transformation is the same as non-private neural networks, since we introduce a precomputation phase to generate dot-product triples. Therefore, to investigate the overhead introduced by MiniONN, we only need to consider the activation functions and pooling operations in a given neural network model. Figure 16 shows the performance of oblivious ReLU, oblivious square, oblivious sigmoid, and oblivious max operations (used in both pooling and maxout activation functions). Both message size and latency grow sublinearly as the number of invocations increases. The experiments are
repeated five times for each point. The standard deviation is below 2.5% for all points.

Model complexity vs. prediction accuracy. The largest contribution to overhead in the online phase are due to activation function usage. We evaluated the performance of our ReLU/CNN/MNIST network (Figure 12) by decreasing the number of neurons in linear layers and the number of channels in convolutional layers, to a fraction of the original value, according to the changes introduced in Figure 17. This effectively reduced the number of activation function instances to the same fraction.

<table>
<thead>
<tr>
<th>α</th>
<th>Accuracy (%)</th>
<th>Overhead (s)</th>
<th>Message size (MB)</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000</td>
<td>5.72</td>
<td>636.6</td>
<td>99.31</td>
<td></td>
</tr>
<tr>
<td>0.975</td>
<td>5.01</td>
<td>557.0</td>
<td>99.27</td>
<td></td>
</tr>
<tr>
<td>0.750</td>
<td>4.29</td>
<td>447.5</td>
<td>99.26</td>
<td></td>
</tr>
<tr>
<td>0.625</td>
<td>3.58</td>
<td>397.9</td>
<td>99.19</td>
<td></td>
</tr>
<tr>
<td>0.500</td>
<td>2.87</td>
<td>317.6</td>
<td>98.96</td>
<td></td>
</tr>
<tr>
<td>0.375</td>
<td>2.15</td>
<td>238.7</td>
<td>98.79</td>
<td></td>
</tr>
<tr>
<td>0.250</td>
<td>1.44 (1.51)</td>
<td>158.4 (159.2)</td>
<td>98.42</td>
<td></td>
</tr>
<tr>
<td>0.188</td>
<td>1.07</td>
<td>119</td>
<td>97.35</td>
<td></td>
</tr>
<tr>
<td>0.125</td>
<td>0.72</td>
<td>79.0</td>
<td>95.72</td>
<td></td>
</tr>
</tbody>
</table>

Table 6 show the estimated latencies and message sizes for different accuracy levels. Thus, if the latency and message size for a particular ONN is perceived as too high, the designer has the option of choosing a suitable point in the accuracy vs. overhead tradeoff. The overhead estimates were approximated, but reasonably accurate. For instance, an actual ReLU/CNN/MNIST model with only 25% ReLU invocations results in 1.51s latency and 159.2MB message sizes, both of which are close to the estimate for α = 0.25 in Table 6.

8 RELATED WORK

Barni et al. [7] made the first attempt to construct oblivious neural networks. They simply have S do linear operations on C’s encrypted data and send the results back to C, who decrypts, applies the non-linear transformations on the plaintexts, and re-encrypts the results before sending them to S for next layer processing. Orlandi et al. [46] noticed that this process leaks significant information about S’s neural network, and proposed a method to obscure the intermediate results. For example, when S needs to know sign(x) from E(pk_C, x), they have S sends a @ E(pk_C, x) to C with a > 0. Obviously, this leaks the sign of x to C. Our work is targeted for the same setting as these works but provides stricter security guarantees (no intermediate results will be leaked) and has significantly better performance.

Gilad-Bachrach et al. [27] proposed CryptoNets based on leveled homomorphic encryption (LHE). They introduced a simple square activation function [27]: f(y) = y^2, because CryptoNets cannot
support commonly used activation functions due to the limitations of LHE. They also used mean pooling instead of max pooling for the same reason, even though the latter is more commonly used. In contrast, MiniONN supports all operations commonly used by neural network designers, does not require changes to how neural networks are trained, and has significantly lower overheads at prediction time. The privacy guarantees to the client are identical. However, while CryptoNets can hide all information about model from clients, MiniONN hides the model values (e.g., weight matrices and bias vectors) while disclosing the number of layers, sizes of weight matrices and the types of operations used in each layer. We argue that this is a justifiable tradeoff for two reasons. First, the performance gain resulting from the tradeoff are truly significant (e.g., 740-fold improvement in online latency). Second, details of a model that are disclosed by MiniONN (like the number of layers and the types of operations) are exactly those that are described in academic and white papers. Model values (like weight matrices and bias vectors in each layer) are usually not disclosed in such literature.

Chabanne et al. [15] also noticed the limited accuracy guarantees of the square function in CryptoNets. They approximated ReLU using a low degree polynomial, and added a normalization layer to make a stable and normal distributed inputs to the activation layer. However, they require a multiplicative depth of 6 in LHE, and they did not provide benchmark results in their paper.

Most of the related works focus on the privacy of training phase (see [4, 5, 30]). For example, Graepel et al. [30] proposed to use training algorithms that can be expressed as low degree polynomials, so that the training phase can be done over encrypted data. Aslett et al. [4, 5] presented ways to train both simple models (e.g., Naive Bayes) as well as more advanced models (e.g., random forests) over encrypted data. The work on differential privacy can also guarantee the privacy in training phase (see [2, 26, 50]). By leveraging Intel SGX combined with several obfuscation algorithms, Ohrimenko et al. [45] proposed a way to enable multiple parties to jointly run a training algorithm while guaranteeing the privacy of their individual datasets.

Recently, in SecureML Mohassel and Zhang proposed a two-server model for privacy-preserving training [43]. Specifically, the data owners distribute their data among two non-colluding servers to train various models including neural networks using secure two-party computation (2PC). While their focus is on training, they also support privacy-preserving predictions. As such their work is closest to ours. Independently of us, they too use a precomputation stage to reduce the overhead during the online prediction phase, support some popular activation functions like ReLU and use approximations where necessary. MiniONN is different from their work in several ways. First, by using the SIMD batch processing technique, MiniONN achieves a significant reduction in the overhead during precomputation without affecting the online phase (Section 6.1). Second, their approximations require changes to how models are trained while the distinguishing characteristic of MiniONN is that it imposes no such requirement.

9 CONCLUSION AND FUTURE WORK

In this paper, we presented MiniONN, which is the first approach that can transform any common neural network into an oblivious form. Our benchmarks show that MiniONN achieves significantly lower response latency and message sizes compared to prior work [27, 43].

We intend to design easy-to-use interfaces that allow developers without any cryptographic background to use MiniONN directly. We also intend to investigate whether our approach is applicable to other machine learning models. As a next step, we plan to apply MiniONN to the neural networks that are being used in production.

REFERENCES
